# CPSC 330
# Applied Machine Learning

## Lecture 7: Linear Models

UBC 2022-23

Instructor: Mathias Lécuyer

## Imports

In [2]:

```
import os
import sys

sys.path.append("../code/.")

import IPython
import ipywidgets as widgets
import matplotlib.pyplot as plt
import mglearn
import numpy as np
import pandas as pd
from IPython.display import HTML, display
from ipywidgets import interact, interactive
from plotting_functions import *
from sklearn.dummy import DummyClassifier
from sklearn.feature_extraction.text import CountVectorizer, TfidfVecto
from sklearn.impute import SimpleImputer
from sklearn.model_selection import cross_val_score, cross_validate, tr
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.compose import make_column_transformer
from utils import *

%matplotlib inline
pd.set_option("display.max_colwidth", 200)
```

## Learning outcomes

From this lecture, students are expected to be able to:

- Explain how `predict` works for linear regression;
- Use `scikit-learn` 's `Ridge` model;
- Demonstrate how the `alpha` hyperparameter of `Ridge` is related to the fundamental tradeoff;
- Explain the difference between linear regression and logistic regression;
- Use `scikit-learn` 's `LogisticRegression` model and `predict_proba` to get probability scores
- Explain the advantages of getting probability scores instead of hard predictions during classification;
- Broadly describe linear SVMs
- Explain how one can interpret model predictions using coefficients learned by a linear model;
- Explain the advantages and limitations of linear classifiers
- Carry out multi-class classification using One-Vs-Rest(All) and One-Vs-One strategies (later, Lecture 17).

# Linear models [video (https://youtu.be/HXd1U2q4VFA)]

**Linear models** is a fundamental and widely used class of models. They are called **linear** because they make a prediction using a **linear function** of the input features.

We will talk about three linear models:

- Linear regression
- Logistic regression
- Linear SVM (brief mention)

## Linear regression

- A very popular statistical model and has a long history.
- Imagine a hypothetical regression problem of predicting the weight of a snake given its length.

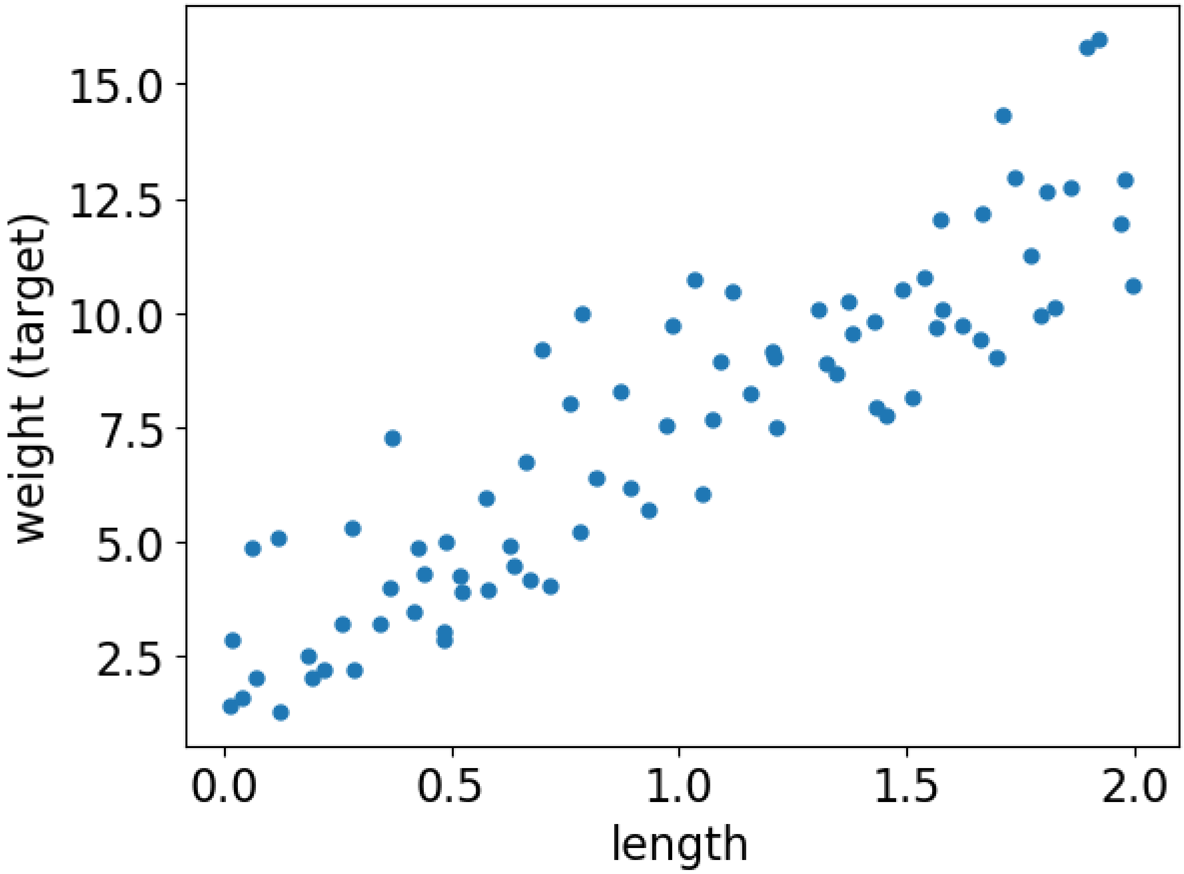
In [3]:
```
np.random.seed(7)
n = 100
X_1 = np.linspace(0, 2, n) + np.random.randn(n) * 0.01
X = pd.DataFrame(X_1[:, None], columns=["length"])

y = abs(np.random.randn(n, 1)) * 3 + X_1[:, None] * 5 + 0.2
y = pd.DataFrame(y, columns=["weight"])
snakes_df = pd.concat([X, y], axis=1)
train_df, test_df = train_test_split(snakes_df, test_size=0.2, random_s

X_train = train_df[["length"]]
y_train = train_df["weight"]
X_test = test_df[["length"]]
y_test = test_df["weight"]
train_df.head()
```

Out[3]:

| | length | weight |
|---|---|---|
| **73** | 1.489130 | 10.507995 |
| **53** | 1.073233 | 7.658047 |
| **80** | 1.622709 | 9.748797 |
| **49** | 0.984653 | 9.731572 |
| **23** | 0.484937 | 3.016555 |

Let's visualize the hypothetical snake data.

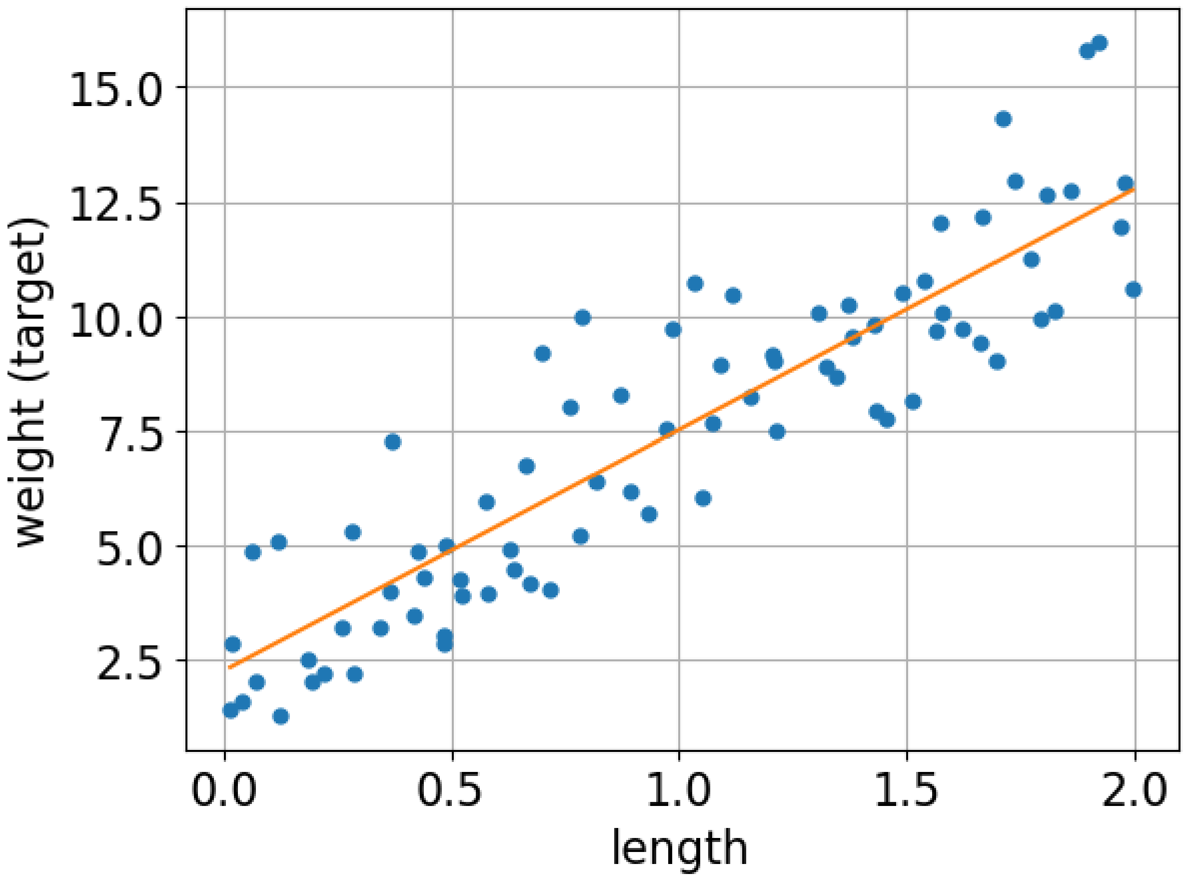
In [4]:
```
plt.plot(X_train, y_train, ".", markersize=10)
plt.xlabel("length")
plt.ylabel("weight (target)");
```




Let's plot a linear regression model on this dataset.

In [5]:
```
grid = np.linspace(min(X_train.to_numpy())[0], max(X_train.to_numpy())[
grid = grid.reshape(-1, 1)
```

In [6]:
```python
from sklearn.linear_model import Ridge

r = Ridge()
r.fit(X_train.to_numpy(), y_train)
plt.plot(X_train, y_train, ".", markersize=10)
plt.plot(grid, r.predict(grid))
plt.grid(True)
plt.xlabel("length")
plt.ylabel("weight (target)");
```




**The orange line is the learned linear model.**

### Prediction of linear regression

- Given a snake length, we can use the model above to predict the target (i.e., the weight of the snake).
- The prediction will be the corresponding weight on the orange line.

In [7]:
```python
snake_length = 0.75
r.predict([[snake_length]])
```

Out[7]:
```
array([6.20683258])
```

**What are we exactly learning?**

- The model above is a line, which can be represented with a slope (i.e., coefficient or weight) and an intercept.
- For the above model, we can access the slope (i.e., coefficient or weight) and the intercept using `coef_` and `intercept_`, respectively.

In [8]:
```
r.coef_  # r is our linear regression object
```
Out[8]:
```
array([5.26370005])
```

In [9]:
```
r.intercept_  # r is our linear regression object
```
Out[9]:
```
2.259057547817185
```

### How are we making predictions?

- Given a feature value $x_1$ and learned coefficient $w_1$ and intercept $b$, we can get the prediction $\hat{y}$ with the following formula:

$$\hat{y} = w_1 x_1 + b$$

In [10]:
```
prediction = snake_length * r.coef_ + r.intercept_
prediction
```
Out[10]:
```
array([6.20683258])
```

In [11]:
```
r.predict([[snake_length]])
```
Out[11]:
```
array([6.20683258])
```

Great! Now we exactly know how the model is making the prediction.

### Generalizing to more features

For more features, the model is a higher dimensional hyperplane and the general prediction formula looks as follows:

$\hat{y} = w_1\ x_1 + \dots + w_d\ x_d + b$

where,

- $(x_1, \dots, x_d)$ are input features
- $(w_1, \dots, w_d)$ are coefficients or weights (learned from the data)
- $b$ is the bias which can be used to offset your hyperplane (learned from the data)

### Example

- Suppose these are the coefficients learned by a linear regression model on a hypothetical housing price prediction dataset.

| Feature | Learned coefficient |
|---|---|
| Bedrooms | 0.20 |
| Bathrooms | 0.11 |
| Square Footage | 0.002 |
| Age | -0.02 |

- Now given a new example, the target will be predicted as follows:

| Bedrooms | Bathrooms | Square Footage | Age |
|---|---|---|---|
| 3 | 2 | 1875 | 66 |

$$\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$$

$$\text{predicted price} = 0.20 \times 3 + 0.11 \times 2 + 0.002 \times 1875 + (-0.02) \times 66 + b$$

When we call `fit` , a coefficient or weight is learned for each feature which tells us the role of that feature in prediction. These coefficients are learned from the training data.

```
In linear models for regression, the model is a line for a single f
eature, a plane for two features, and a hyperplane for higher dimen
sions. We are not yet ready to discuss how does linear regression l
earn these coefficients and intercept.
```

### Ridge

- `scikit-learn` has a model called `LinearRegression` for linear regression.
- But if we use this "vanilla" version of linear regression, it may result in large coefficients and unexpected results.
- So instead of using `LinearRegression` , we will always use another linear model called `Ridge` , which is a linear regression model with a complexity hyperparameter `alpha` .
- If you want to know more about how Ridge regularization works, we recommend this tutorial: [https://www.analyticsvidhya.com/blog/2016/01/ridge-lasso-regression-python-complete-tutorial/ (https://www.analyticsvidhya.com/blog/2016/01/ridge-lasso-regression-python-complete-tutorial/)](https://www.analyticsvidhya.com/blog/2016/01/ridge-lasso-regression-python-complete-tutorial/)

In [12]:
```
from sklearn.linear_model import LinearRegression  # DO NOT USE IT
from sklearn.linear_model import Ridge  # USE THIS INSTEAD
```

#### Data

In [13]:
```
housing_df = pd.read_csv("../data/housing.csv")
train_df, test_df = train_test_split(housing_df, test_size=0.1, random_

train_df = train_df.assign(
    rooms_per_household=train_df["total_rooms"] / train_df["households"
)
test_df = test_df.assign(
    rooms_per_household=test_df["total_rooms"] / test_df["households"]
)

train_df = train_df.assign(
    bedrooms_per_household=train_df["total_bedrooms"] / train_df["house
)
test_df = test_df.assign(
    bedrooms_per_household=test_df["total_bedrooms"] / test_df["househo
)

train_df = train_df.assign(
    population_per_household=train_df["population"] / train_df["househo
)
test_df = test_df.assign(
    population_per_household=test_df["population"] / test_df["household
)

X_train = train_df.drop(columns=["median_house_value"])
y_train = train_df["median_house_value"]

X_test = test_df.drop(columns=["median_house_value"])
y_test = test_df["median_house_value"]

numeric_features = [
    "housing_median_age",
    "population",
    "households",
    "median_income",
    "rooms_per_household",
    "bedrooms_per_household",
    "population_per_household",
]
categorical_features = ["ocean_proximity"]
drop_features = [
    "longitude",
    "latitude",
    "total_rooms",
    "total_bedrooms",
]

ct = make_column_transformer(
    (
        make_pipeline(SimpleImputer(strategy="median"), StandardScaler(
        numeric_features,
    ),
    (
        OneHotEncoder(handle_unknown="ignore"),
        categorical_features,
    ),
    ("drop", drop_features),
```

```
)
```

In [14]:

```
pipe = make_pipeline(ct, Ridge())
scores = cross_validate(pipe, X_train, y_train, return_train_score=True
pd.DataFrame(scores)
```

Out[14]:

| | fit_time | score_time | test_score | train_score |
|---|---|---|---|---|
| 0 | 0.038253 | 0.007985 | 0.622840 | 0.634925 |
| 1 | 0.031309 | 0.006604 | 0.651388 | 0.627447 |
| 2 | 0.034430 | 0.006995 | 0.635000 | 0.632330 |
| 3 | 0.033828 | 0.007207 | 0.622482 | 0.635518 |
| 4 | 0.035256 | 0.008641 | 0.595064 | 0.635765 |

### Hyperparameter `alpha` of `Ridge`

- Ridge has hyperparameters just like the rest of the models we learned.
- The alpha hyperparameter is what makes `Ridge` different from vanilla `LinearRegression`.
- Similar to the other hyperparameters that we saw, `alpha` controls the fundamental tradeoff.

```
If we set alpha=0 that is the same as using LinearRegression.
```

Let's examine the effect of `alpha` on the fundamental tradeoff.

In [15]:

```
scores_dict = {
    "alpha": 10.0 ** np.arange(-2, 6, 1),
    "mean_train_scores": list(),
    "mean_cv_scores": list(),
}
for alpha in scores_dict["alpha"]:
    pipe_ridge = make_pipeline(ct, Ridge(alpha=alpha))
    scores = cross_validate(pipe_ridge, X_train, y_train, return_train_
    scores_dict["mean_train_scores"].append(scores["train_score"].mean(
    scores_dict["mean_cv_scores"].append(scores["test_score"].mean())

results_df = pd.DataFrame(scores_dict)
```

In [16]:
```
results_df
```

Out[16]:

| | alpha | mean_train_scores | mean_cv_scores |
|---|---|---|---|
| **0** | 0.01 | 0.633217 | 0.625408 |
| **1** | 0.10 | 0.633216 | 0.625405 |
| **2** | 1.00 | 0.633197 | 0.625355 |
| **3** | 10.00 | 0.632909 | 0.624734 |
| **4** | 100.00 | 0.632352 | 0.620946 |
| **5** | 1000.00 | 0.621155 | 0.578949 |
| **6** | 10000.00 | 0.490031 | 0.438445 |
| **7** | 100000.00 | 0.140880 | 0.137042 |

Here we do not really see overfitting but in general,

- larger `alpha` → likely to underfit
- smaller `alpha` → likely to overfit

### Coefficients and intercept

The model learns

- coefficients associated with each feature
- the intercept or bias

Let's examine the coefficients learned by the model.

In [17]:
```
pipe_ridge = make_pipeline(ct, Ridge(alpha=10.0))
pipe_ridge.fit(X_train, y_train)
coeffs = pipe_ridge.named_steps["ridge"].coef_
```

In [18]:
```python
column_names = numeric_features + list(
    ct.named_transformers_["onehotencoder"].get_feature_names_out(
        categorical_features
    )
)

pd.DataFrame(data=coeffs, index=column_names, columns=["Coefficients"])
```

Out[18]:

| | Coefficients |
|---|---|
| **housing_median_age** | 14658.019489 |
| **population** | -43890.006653 |
| **households** | 49138.464132 |
| **median_income** | 78819.238154 |
| **rooms_per_household** | -15062.467094 |
| **bedrooms_per_household** | 17543.750856 |
| **population_per_household** | 625.875537 |
| **ocean_proximity_<1H OCEAN** | -1660.944525 |
| **ocean_proximity_INLAND** | -68031.703729 |
| **ocean_proximity_ISLAND** | 56468.596208 |
| **ocean_proximity_NEAR BAY** | 1529.230447 |
| **ocean_proximity_NEAR OCEAN** | 11694.821599 |

- The model also learns an intercept (bias).
- For each prediction, we are adding this amount irrespective of the feature values.

In [19]:
```python
pipe_ridge.named_steps["ridge"].intercept_
```

Out[19]:
```
227002.23569246812
```

Can we use this information to interpret model predictions?

## ❓❓ Questions for you

**True/False: `Ridge`**

1. Increasing the hyperparameter `alpha` of `Ridge` is likely to decrease model complexity.
2. `Ridge` can be used with datasets that have multiple features.
3. With Ridge, we learn one coefficient per training example.
4. If you train a linear regression model on a 2-dimensional problem (2 features), the model will be a two dimensional plane.

## Interpretation of coefficients

- One of the main advantages of linear models is that they are relatively easy to interpret.
- We have one coefficient per feature which kind of describes the role of the feature in the prediction according to the model.

There are two pieces of information in the coefficients based on

- Sign
- Magnitude

### Sign of the coefficients

In [20]:
```
pd.DataFrame(data=coeffs, index=column_names, columns=["Coefficients"])
```

Out[20]:

| | Coefficients |
|---|---|
| **housing_median_age** | 14658.019489 |
| **population** | -43890.006653 |
| **households** | 49138.464132 |
| **median_income** | 78819.238154 |
| **rooms_per_household** | -15062.467094 |
| **bedrooms_per_household** | 17543.750856 |
| **population_per_household** | 625.875537 |
| **ocean_proximity_<1H OCEAN** | -1660.944525 |
| **ocean_proximity_INLAND** | -68031.703729 |
| **ocean_proximity_ISLAND** | 56468.596208 |
| **ocean_proximity_NEAR BAY** | 1529.230447 |
| **ocean_proximity_NEAR OCEAN** | 11694.821599 |

#### Magnitude of the coefficients

- Bigger magnitude → bigger impact on the prediction
- In the example below, both RM and AGE have a positive impact on the prediction but RM would have a bigger positive impact because it's feature value is going to be multiplied by a number with a bigger magnitude.

- Similarly both LSAT and NOX have a negative impact on the prediction but LSAT would have a bigger negative impact because it's going to be multiplied by a number with a bigger magnitude.

In [21]:

```
data = {
    "coefficient": pipe_ridge.named_steps["ridge"].coef_.tolist(),
    "magnitude": np.absolute(pipe_ridge.named_steps["ridge"].coef_.toli
}
coef_df = pd.DataFrame(data, index=column_names).sort_values(
    "magnitude", ascending=False
)
coef_df
```

Out[21]:

| | coefficient | magnitude |
|---|---|---|
| **median_income** | 78819.238154 | 78819.238154 |
| **ocean_proximity_INLAND** | -68031.703729 | 68031.703729 |
| **ocean_proximity_ISLAND** | 56468.596208 | 56468.596208 |
| **households** | 49138.464132 | 49138.464132 |
| **population** | -43890.006653 | 43890.006653 |
| **bedrooms_per_household** | 17543.750856 | 17543.750856 |
| **rooms_per_household** | -15062.467094 | 15062.467094 |
| **housing_median_age** | 14658.019489 | 14658.019489 |
| **ocean_proximity_NEAR OCEAN** | 11694.821599 | 11694.821599 |
| **ocean_proximity_<1H OCEAN** | -1660.944525 | 1660.944525 |
| **ocean_proximity_NEAR BAY** | 1529.230447 | 1529.230447 |
| **population_per_household** | 625.875537 | 625.875537 |

## Importance of scaling

- When you are interpreting the model coefficients, scaling is crucial.
- If you do not scale the data, features with smaller magnitude are going to get coefficients with bigger magnitude whereas features with bigger scale are going to get coefficients with smaller magnitude.
- That said, when you scale the data, feature values become hard to interpret for humans!

```
Take these coefficients with a grain of salt. They might not always
match your intuitions.
```

## ❓❓ Questions for you

**True/False**

1. Suppose you have trained a linear model on an unscaled data. The coefficients of the linear model have the following interpretation: if coefficient $j$ is large, that means a change in feature $j$ has a large impact on the prediction.
2. Suppose the scaled feature value of `population` above is negative. The prediction will still be inversely proportional to `population`; as `population` gets **bigger**, the median house value gets **smaller**.

**Questions for breakout room discussion**

- Discuss the importance of scaling when interpreting linear regression coefficients.
- What might be the meaning of complex vs simpler model in case of linear regression?

# Logistic regression [video (https://youtu.be/56L5z_t22qE)]

## Logistic regression intuition

- A linear model for **classification**.
- Similar to linear regression, it learns weights associated with each feature and the bias.
- It applies a **threshold** on the raw output to decide whether the class is positive or negative.
- In this lecture we will focus on the following aspects of logistic regression.
  - `predict`, `predict_proba`
  - how to use learned coefficients to interpret the model

## Motivating example

- Consider the problem of predicting sentiment expressed in movie reviews.

**Training data for the motivating example**

> Review 1: This movie was **excellent**! The performances were oscar-worthy! 👍
>
> Review 2: What a **boring** movie! I almost fell asleep twice while watching it. 👎
>
> Review 3: I enjoyed the movie. **Excellent**! 👍

- Targets: positive 👍 and negative 👎
- Features: words (e.g., *excellent*, *flawless*, *boring*)

#### Learned coefficients associated with all features

- Suppose our vocabulary contains only the following 7 words.
- A linear classifier learns **weights** or **coefficients** associated with the features (words in this example).
- Let's ignore bias for a bit.

| Word | Coefficient |
|---|---|
| excellent | 1.93 |
| disappointment | -2.20 |
| flawless | 1.43 |
| boring | -1.40 |
| unwatchable | -2.04 |
| incoherent | -1.86 |
| subtle | 1.30 |

#### Predicting with learned weights

- Use these learned coefficients to make predictions. For example, consider the following review $x_i$.

> It got a bit **boring** at times but the direction was **excellent** and the acting was **flawless**.

- Feature vector for $x_i$: [1, 0, 1, 1, 0, 0, 0]

| Word | Coefficient |
|---|---|
| excellent | 1.93 |
| disappointment | -2.20 |
| flawless | 1.43 |
| boring | -1.40 |
| unwatchable | -2.04 |
| incoherent | -1.86 |
| subtle | 1.30 |

- $score(x_i) =$ coefficient(*boring*) $\times 1$ + coefficient(*excellent*) $\times 1$ + coefficient(*flawless*) $\times 1$ = $-1.40 + 1.93 + 1.43 = 1.96$
- $1.96 > 0$ so predict the review as positive 👍.

In [22]:
```
x = ["boring=1", "excellent=1", "flawless=1"]
w = [-1.40, 1.93, 1.43]
display(plot_logistic_regression(x, w))
```

```
Weighted sum of the input features = 1.960 y_hat = pos

<graphviz.graphs.Digraph at 0x18a688520>
```

- So the prediction is based on the weighted sum of the input features.
- Some feature are pulling the prediction towards positive sentiment and some are pulling it towards negative sentiment.
- If the coefficient of *boring* had a bigger magnitude or *excellent* and *flawless* had smaller magnitudes, we would have predicted "neg".

In [23]:
```
def f(w_0):
    x = ["boring=1", "excellent=1", "flawless=1"]
    w = [-1.40, 1.93, 1.43]
    w[0] = w_0
    print(w)
    display(plot_logistic_regression(x, w))
```

In [24]:
```
interactive(
    f,
    w_0=widgets.FloatSlider(min=-6, max=2, step=0.5, value=-1.40),
)
```

Out[24]:
```
interactive(children=(FloatSlider(value=-1.4, description='w_0', max=2.0,
min=-6.0, step=0.5), Output()), _dom…
```

In our case, for values for the coefficient of *boring* < -3.36, the prediction would be negative.

So a linear classifier is a linear function of the input `X`, followed by a threshold.

$$
\begin{aligned}
z =& w_1 x_1 + \dots + w_d x_d + b\\
=& w^T x + b
\end{aligned}
$$

$$
\hat{y} = \begin{cases} 1, & \text{if } z \geq r\\ -1, & \text{if } z < r \end{cases}
$$

**Components of a linear classifier**

1. input features ($x_1, \dots, x_d$)
2. coefficients (weights) ($w_1, \dots, w_d$)
3. bias ($b$ or $w_0$) (can be used to offset your hyperplane)
4. threshold ($r$)

In our example before, we assumed $r = 0$ and $b = 0$.

## Logistic regression on the cities data

In [25]:

```
cities_df = pd.read_csv("../data/canada_usa_cities.csv")
train_df, test_df = train_test_split(cities_df, test_size=0.2, random_s
X_train, y_train = train_df.drop(columns=["country"], axis=1), train_df
X_test, y_test = test_df.drop(columns=["country"], axis=1), test_df["co

train_df.head()
```

Out[25]:

| | longitude | latitude | country |
|---|---|---|---|
| **160** | -76.4813 | 44.2307 | Canada |
| **127** | -81.2496 | 42.9837 | Canada |
| **169** | -66.0580 | 45.2788 | Canada |
| **188** | -73.2533 | 45.3057 | Canada |
| **187** | -67.9245 | 47.1652 | Canada |

Let's first try `DummyClassifier` on the cities data.

In [26]:
```
dummy = DummyClassifier()
scores = cross_validate(dummy, X_train, y_train, return_train_score=Tru
pd.DataFrame(scores)
```

Out[26]:

| | fit_time | score_time | test_score | train_score |
|---|---|---|---|---|
| **0** | 0.001376 | 0.002050 | 0.588235 | 0.601504 |
| **1** | 0.000801 | 0.000412 | 0.588235 | 0.601504 |
| **2** | 0.001103 | 0.000663 | 0.606061 | 0.597015 |
| **3** | 0.000771 | 0.000396 | 0.606061 | 0.597015 |
| **4** | 0.000691 | 0.000353 | 0.606061 | 0.597015 |

Now let's try `LogisticRegression`

In [27]:
```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
scores = cross_validate(lr, X_train, y_train, return_train_score=True)
pd.DataFrame(scores)
```

Out[27]:

| | fit_time | score_time | test_score | train_score |
|---|---|---|---|---|
| **0** | 0.014193 | 0.001574 | 0.852941 | 0.827068 |
| **1** | 0.007429 | 0.001624 | 0.823529 | 0.827068 |
| **2** | 0.006945 | 0.001767 | 0.696970 | 0.858209 |
| **3** | 0.008119 | 0.001420 | 0.787879 | 0.843284 |
| **4** | 0.006647 | 0.001773 | 0.939394 | 0.805970 |

Logistic regression seems to be doing better than dummy classifier. But note that there is a lot of variation in the scores.

## Accessing learned parameters

- Recall that logistic regression learns the weights $w$ and bias or intercept $b$.
- How to access these weights?
  - Similar to `Ridge` , we can access the weights and intercept using `coef_` and `intercept_` attribute of the `LogisticRegression` object, respectively.

In [28]:
```
lr = LogisticRegression()
lr.fit(X_train.to_numpy(), y_train)
print("Model weights: %s" % (lr.coef_))  # these are the learned weight
print("Model intercept: %s" % (lr.intercept_))  # this is the bias term
data = {"features": X_train.columns, "coefficients": lr.coef_[0]}
pd.DataFrame(data)
```

```
Model weights: [[-0.04108149 -0.33683126]]
Model intercept: [10.8869838]
```

Out[28]:

| | features | coefficients |
|---|---|---|
| **0** | longitude | -0.041081 |
| **1** | latitude | -0.336831 |

- Both negative weights
- The weight of latitude is larger in magnitude.
- This makes sense because Canada as a country lies above the USA and so we expect latitude values to contribute more to a prediction than longitude.

## Prediction with learned parameters

Let's predict target of a test example.

In [29]:
```
example = X_test.iloc[0, :]
example
```

Out[29]:
```
longitude   -64.8001
latitude     46.0980
Name: 172, dtype: float64
```

### Raw scores

- Calculate the raw score as: `y_hat = np.dot(w, x) + b`

In [30]:
```
(
    np.dot(
        example.to_numpy(),
        lr.coef_.reshape(
            2,
        ),
    )
    + lr.intercept_
)
```

Out[30]:
```
array([-1.97817876])
```

- Apply the threshold to the raw score.
- Since the prediction is < 0, predict "negative".

- What is a "negative" class in our context?
- With logistic regression, the model randomly assigns one of the classes as a positive class and the other as negative.
  - Usually it would alphabetically order the target and pick the first one as negative and second one as the positive class.

- The `classes_` attribute tells us which class is considered negative and which one is considered positive. - In this case, Canada is the negative class and USA is a positive class.

In [31]:
```
lr.classes_
```
Out[31]:
```
array(['Canada', 'USA'], dtype=object)
```

- So based on the negative score above (-1.978), we would predict Canada.
- Let's check the prediction given by the model.

In [32]:
```
lr.predict([example])
```
Out[32]:
```
array(['Canada'], dtype=object)
```

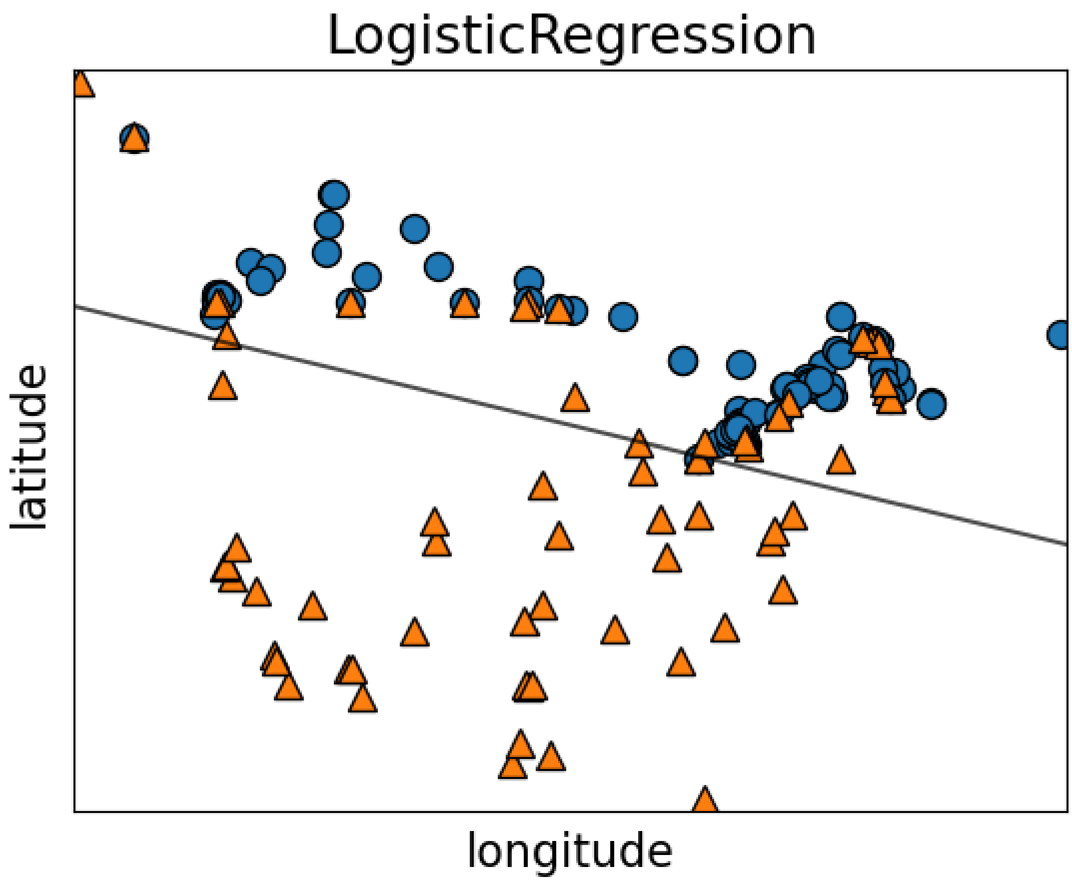
Great! The predictions match! We exactly know how the model is making predictions.

## Decision boundary of logistic regression

- The decision boundary of logistic regression is a **hyperplane** dividing the feature space in half.

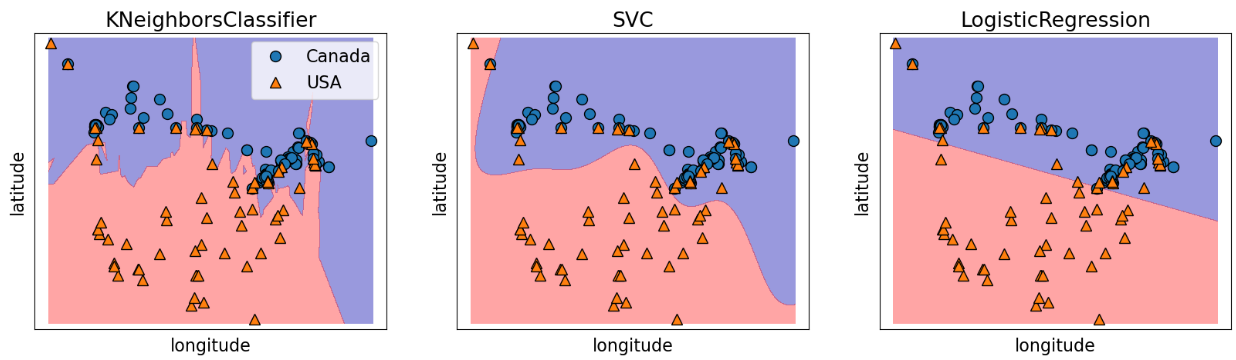
In [33]:

```
lr = LogisticRegression()
lr.fit(X_train.to_numpy(), y_train)
mglearn.discrete_scatter(X_train.iloc[:, 0], X_train.iloc[:, 1], y_trai
mglearn.plots.plot_2d_separator(lr, X_train.to_numpy(), fill=False, eps
plt.title(lr.__class__.__name__)
plt.xlabel("longitude")
plt.ylabel("latitude");
```




- For $d = 2$, the decision boundary is a line (1-dimensional)
- For $d = 3$, the decision boundary is a plane (2-dimensional)
- For $d > 3$, the decision boundary is a $d - 1$-dimensional hyperplane

In [34]:
```
fig, axes = plt.subplots(1, 3, figsize=(20, 5))
for model, ax in zip(
    [KNeighborsClassifier(), SVC(gamma=0.01), LogisticRegression()], ax
):
    clf = model.fit(X_train.to_numpy(), y_train)
    mglearn.plots.plot_2d_separator(
        clf, X_train.to_numpy(), fill=True, eps=0.5, ax=ax, alpha=0.4
    )
    mglearn.discrete_scatter(X_train.iloc[:, 0], X_train.iloc[:, 1], y_
    ax.set_title(clf.__class__.__name__)
    ax.set_xlabel("longitude")
    ax.set_ylabel("latitude")
axes[0].legend();
```




- Notice a linear decision boundary (a line in our case).
- Compare it with KNN or SVM RBF decision boundaries.

## Main hyperparameter of logistic regression

- `C` is the main hyperparameter which controls the fundamental trade-off.
- We won't really talk about the interpretation of this hyperparameter right now.
- At a high level, the interpretation is similar to `C` of SVM RBF
  - smaller `C` $\rightarrow$ might lead to underfitting
  - bigger `C` $\rightarrow$ might lead to overfitting

In [35]:

```
scores_dict = {
    "C": 10.0 ** np.arange(-4, 6, 1),
    "mean_train_scores": list(),
    "mean_cv_scores": list(),
}
for C in scores_dict["C"]:
    lr = LogisticRegression(C=C)
    scores = cross_validate(lr, X_train, y_train, return_train_score=Tr
    scores_dict["mean_train_scores"].append(scores["train_score"].mean(
    scores_dict["mean_cv_scores"].append(scores["test_score"].mean())

results_df = pd.DataFrame(scores_dict)
results_df
```

Out[35]:

| | C | mean_train_scores | mean_cv_scores |
|---|---|---|---|
| **0** | 0.0001 | 0.664707 | 0.658645 |
| **1** | 0.0010 | 0.784424 | 0.790731 |
| **2** | 0.0100 | 0.827842 | 0.826203 |
| **3** | 0.1000 | 0.832320 | 0.820143 |
| **4** | 1.0000 | 0.832320 | 0.820143 |
| **5** | 10.0000 | 0.832320 | 0.820143 |
| **6** | 100.0000 | 0.832320 | 0.820143 |
| **7** | 1000.0000 | 0.832320 | 0.820143 |
| **8** | 10000.0000 | 0.832320 | 0.820143 |
| **9** | 100000.0000 | 0.832320 | 0.820143 |

## Predicting probability scores [video (https://youtu.be/_OAK5KiGLg0)]

### `predict_proba`

- So far in the context of classification problems, we focused on getting "hard" predictions.
- Very often it's useful to know "soft" predictions, i.e., how confident the model is with a given prediction.
- For most of the `scikit-learn` classification models we can access this confidence score or probability score using a method called `predict_proba`.

Let's look at probability scores of logistic regression model for our test example.

In [36]:
```
example
```

Out[36]:
```
longitude   -64.8001
latitude     46.0980
Name: 172, dtype: float64
```

In [37]:
```
lr = LogisticRegression()
lr.fit(X_train.to_numpy(), y_train)
lr.predict([example])  # hard prediction
```

Out[37]:
```
array(['Canada'], dtype=object)
```

In [38]:
```
lr.predict_proba([example])  # soft prediction
```

Out[38]:
```
array([[0.87848688, 0.12151312]])
```

- The output of `predict_proba` is the probability of each class.
- In binary classification, we get probabilities associated with both classes (even though this information is redundant).
- The first entry is the estimated probability of the first class and the second entry is the estimated probability of the second class from `model.classes_`.

In [39]:
```
lr.classes_
```

Out[39]:
```
array(['Canada', 'USA'], dtype=object)
```

- Because it's a probability, the sum of the entries for both classes should always sum to 1.
- Since the probabilities for the two classes sum to 1, exactly one of the classes will have a score >=0.5, which is going to be our predicted class.

**How does logistic regression calculate these probabilities?**

- The weighted sum $w_1 x_1 + \dots + w_d x_d + b$ gives us "raw model output".
- For linear regression this would have been the prediction.
- For logistic regression, you check the **sign** of this value.
  - If positive (or 0), predict $+1$; if negative, predict $-1$.
  - These are "hard predictions".

- You can also have "soft predictions", aka **predicted probabilities**.
  - To convert the raw model output into probabilities, instead of taking the sign, we apply the **sigmoid**.

**The sigmoid function**

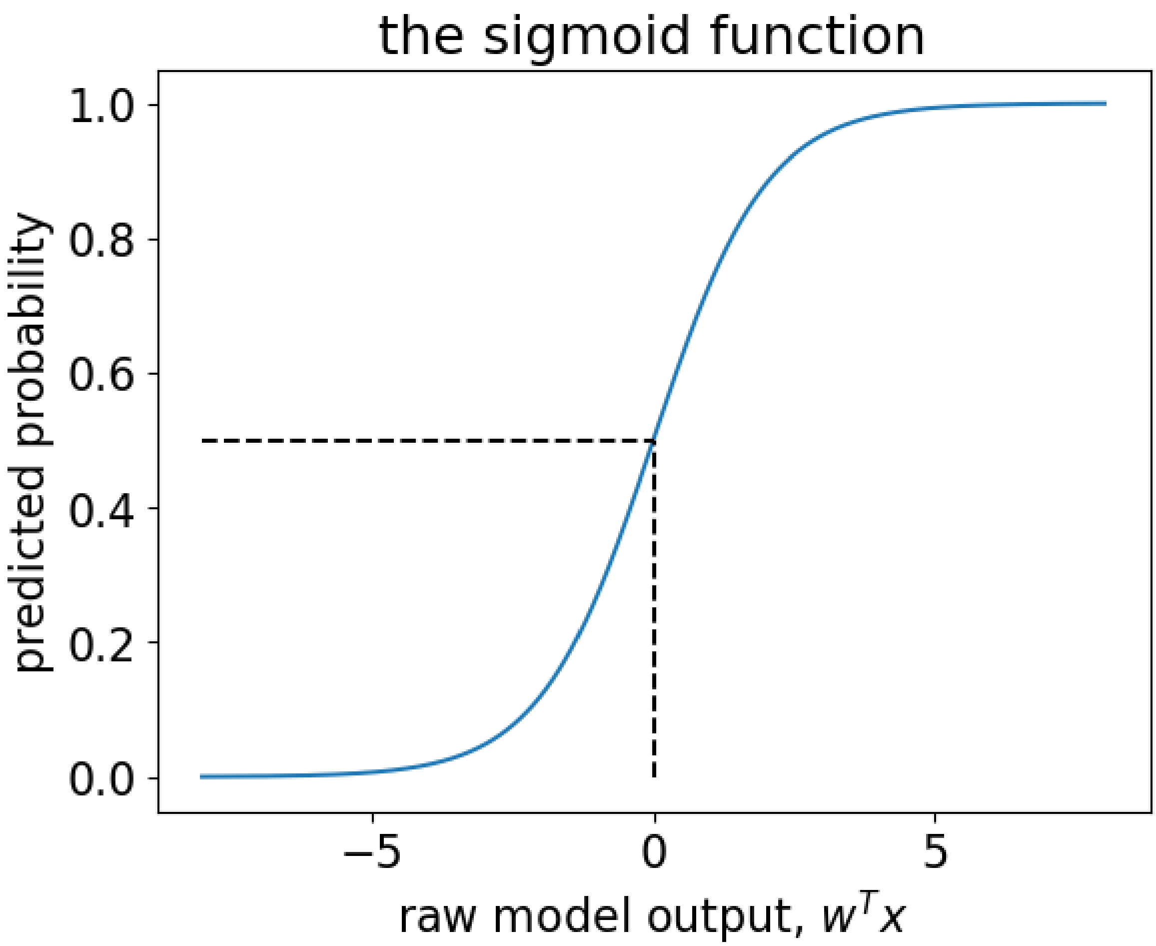
- The sigmoid function "squashes" the raw model output from any number to the range $[0, 1]$ using the following formula, where $x$ is the raw model output.

$$\frac{1}{1 + e^{-x}}$$

- Then we can interpret the output as probabilities.

In [40]:

```
sigmoid = lambda x: 1 / (1 + np.exp(-x))
raw_model_output = np.linspace(-8, 8, 1000)
plt.plot(raw_model_output, sigmoid(raw_model_output))
plt.plot([0, 0], [0, 0.5], "--k")
plt.plot([-8, 0], [0.5, 0.5], "--k")
plt.xlabel("raw model output, $w^Tx$")
plt.ylabel("predicted probability")
plt.title("the sigmoid function");
```




- Recall our hard predictions that check the sign of $w^T x$, or, in other words, whether or not it is $\geq 0$.
  - The threshold $w^T x = 0$ corresponds to $p = 0.5$.
  - In other words, if our predicted probability is $\geq 0.5$ then our hard prediction is $+1$.

Let's get the probability score by calling sigmoid on the raw model output for our test example.

In [41]:
```
sigmoid(
    np.dot(
        example.to_numpy(),
        lr.coef_.reshape(
            2,
        ),
    )
    + lr.intercept_
)
```

Out[41]:
```
array([0.12151312])
```

This is the probability score of the positive class, which is USA.

In [42]:
```
lr.predict_proba([example])
```

Out[42]:
```
array([[0.87848688, 0.12151312]])
```

With `predict_proba` , we get the same probability score for USA!!

- Let's visualize probability scores for some examples.

In [43]:
```
data_dict = {
    "y": y_train.iloc[:12],
    "y_hat": lr.predict(X_train.iloc[:12].to_numpy()).tolist(),
    "probabilities": lr.predict_proba(X_train.iloc[:12].to_numpy()).tol
}
```

In [44]:
```
pd.DataFrame(data_dict)
```

Out[44]:

| | y | y_hat | probabilities |
|---|---|---|---|
| **160** | Canada | Canada | [0.7046068097086481, 0.2953931902913519] |
| **127** | Canada | Canada | [0.563016906204013, 0.436983093795987] |
| **169** | Canada | Canada | [0.8389680973255864, 0.16103190267441364] |
| **188** | Canada | Canada | [0.7964150775404333, 0.20358492245956678] |
| **187** | Canada | Canada | [0.9010806652340972, 0.0989193347659027] |
| **192** | Canada | Canada | [0.7753006388010791, 0.2246993611989209] |
| **62** | USA | USA | [0.03074070460652778, 0.9692592953934722] |
| **141** | Canada | Canada | [0.6880304799160918, 0.3119695200839082] |
| **183** | Canada | Canada | [0.7891358587234145, 0.21086414127658554] |
| **37** | USA | USA | [0.006546969753885357, 0.9934530302461146] |
| **50** | USA | USA | [0.2787419584843098, 0.7212580415156902] |
| **89** | Canada | Canada | [0.8388877146644942, 0.1611122853355058] |

The actual `y` and `y_hat` match in most of the cases but in some cases the model is more confident about the prediction than others.

#### Least confident cases

Let's examine some cases where the model is least confident about the prediction.

In [45]:
```
least_confident_X = X_train.loc[[127, 141]]
least_confident_X
```

Out[45]:

| | longitude | latitude |
|---|---|---|
| **127** | -81.2496 | 42.9837 |
| **141** | -79.6902 | 44.3893 |

In [46]:
```
least_confident_y = y_train.loc[[127, 141]]
least_confident_y
```

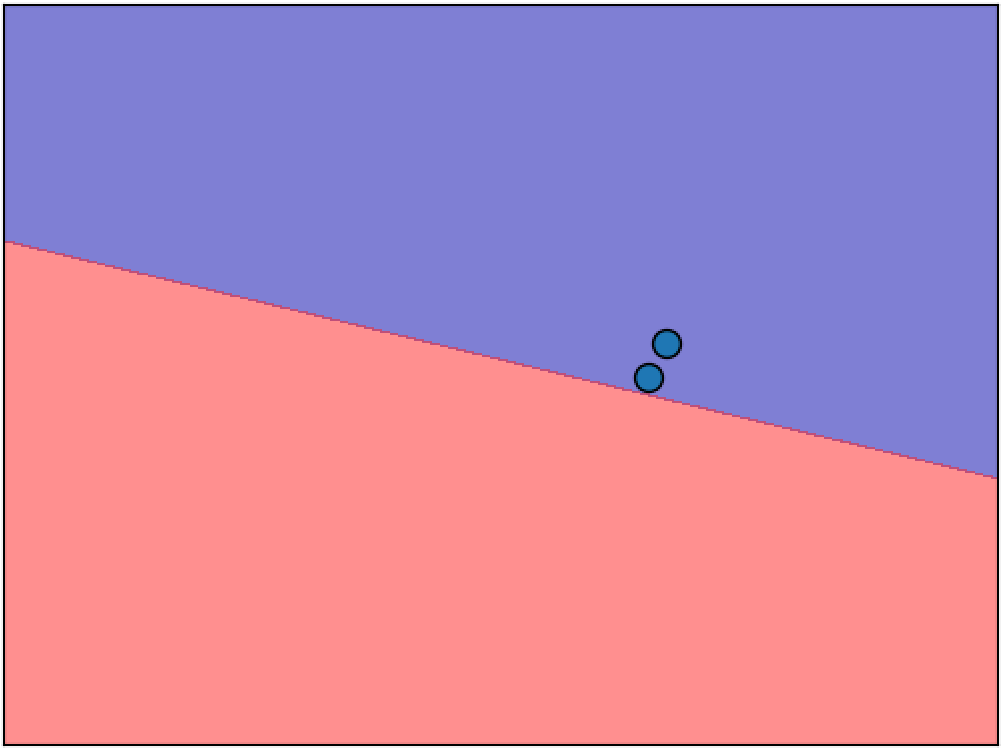
Out[46]:
```
127    Canada
141    Canada
Name: country, dtype: object
```

In [47]:
```
probs = lr.predict_proba(least_confident_X.to_numpy())

data_dict = {
    "y": least_confident_y,
    "y_hat": lr.predict(least_confident_X.to_numpy()).tolist(),
    "probability score (Canada)": probs[:, 0],
    "probability score (USA)": probs[:, 1],
}
pd.DataFrame(data_dict)
```

Out[47]:

| | y | y_hat | probability score (Canada) | probability score (USA) |
|---|---|---|---|---|
| **127** | Canada | Canada | 0.563017 | 0.436983 |
| **141** | Canada | Canada | 0.688030 | 0.311970 |

In [48]:
```
mglearn.discrete_scatter(
    least_confident_X.iloc[:, 0],
    least_confident_X.iloc[:, 1],
    least_confident_y,
    markers="o",
)
mglearn.plots.plot_2d_separator(lr, X_train.to_numpy(), fill=True, eps=
```



The points are close to the decision boundary which makes sense.

#### Most confident cases

Let's examine some cases where the model is most confident about the prediction.

In [49]:
```
most_confident_X = X_train.loc[[37, 165]]
most_confident_X
```

Out[49]:

| | longitude | latitude |
|---|---|---|
| **37** | -98.4951 | 29.4246 |
| **165** | -52.7151 | 47.5617 |

In [50]:
```
most_confident_y = y_train.loc[[37, 165]]
most_confident_y
```

Out[50]:
```
37        USA
165    Canada
Name: country, dtype: object
```

In [51]:
```python
probs = lr.predict_proba(most_confident_X.to_numpy())

data_dict = {
    "y": most_confident_y,
    "y_hat": lr.predict(most_confident_X.to_numpy()).tolist(),
    "probability score (Canada)": probs[:, 0],
    "probability score (USA)": probs[:, 1],
}
pd.DataFrame(data_dict)
```

Out[51]:

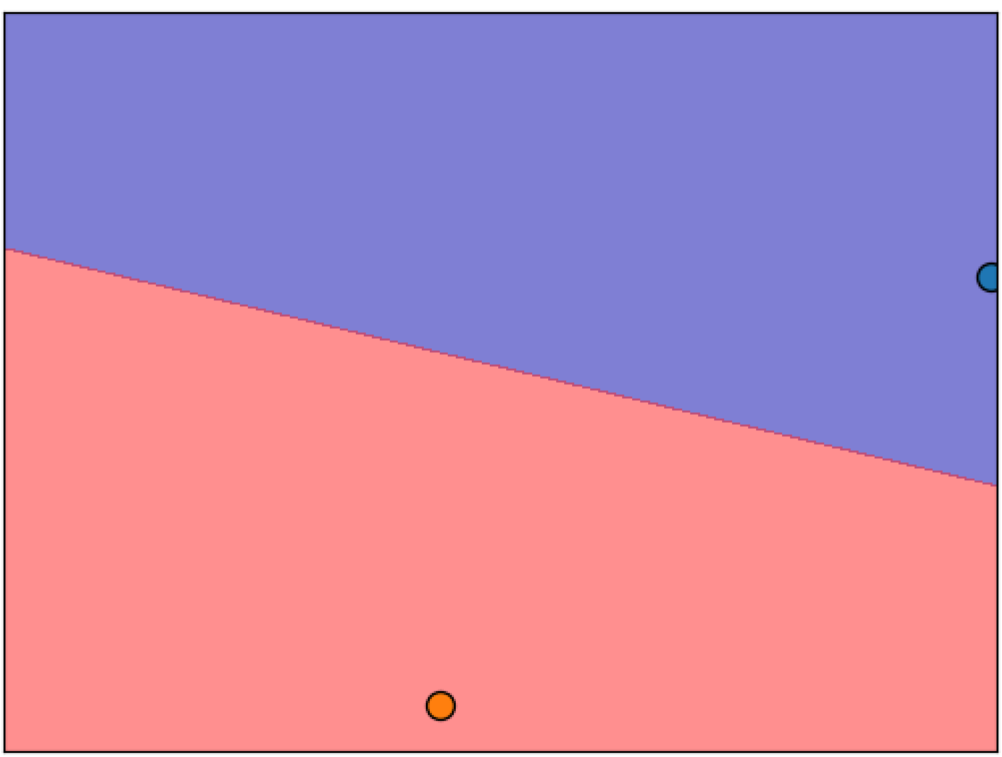
| | y | y_hat | probability score (Canada) | probability score (USA) |
|---|---|---|---|---|
| **37** | USA | USA | 0.006547 | 0.993453 |
| **165** | Canada | Canada | 0.951092 | 0.048908 |

In [52]:
```python
most_confident_X
```

Out[52]:

| | longitude | latitude |
|---|---|---|
| **37** | -98.4951 | 29.4246 |
| **165** | -52.7151 | 47.5617 |

In [53]:
```python
mglearn.discrete_scatter(
    most_confident_X.iloc[:, 0],
    most_confident_X.iloc[:, 1],
    most_confident_y,
    markers="o",
)
mglearn.plots.plot_2d_separator(lr, X_train.to_numpy(), fill=True, eps=
```



The points are far away from the decision boundary which makes sense.

### Over confident cases

Let's examine some cases where the model is confident about the prediction but the prediction is wrong.

In [54]:
```
over_confident_X = X_train.loc[[0, 1]]
over_confident_X
```

Out[54]:

| | longitude | latitude |
|---|---|---|
| **0** | -130.0437 | 55.9773 |
| **1** | -134.4197 | 58.3019 |

In [55]:
```
over_confident_y = y_train.loc[[0, 1]]
over_confident_y
```

Out[55]:
```
0    USA
1    USA
Name: country, dtype: object
```

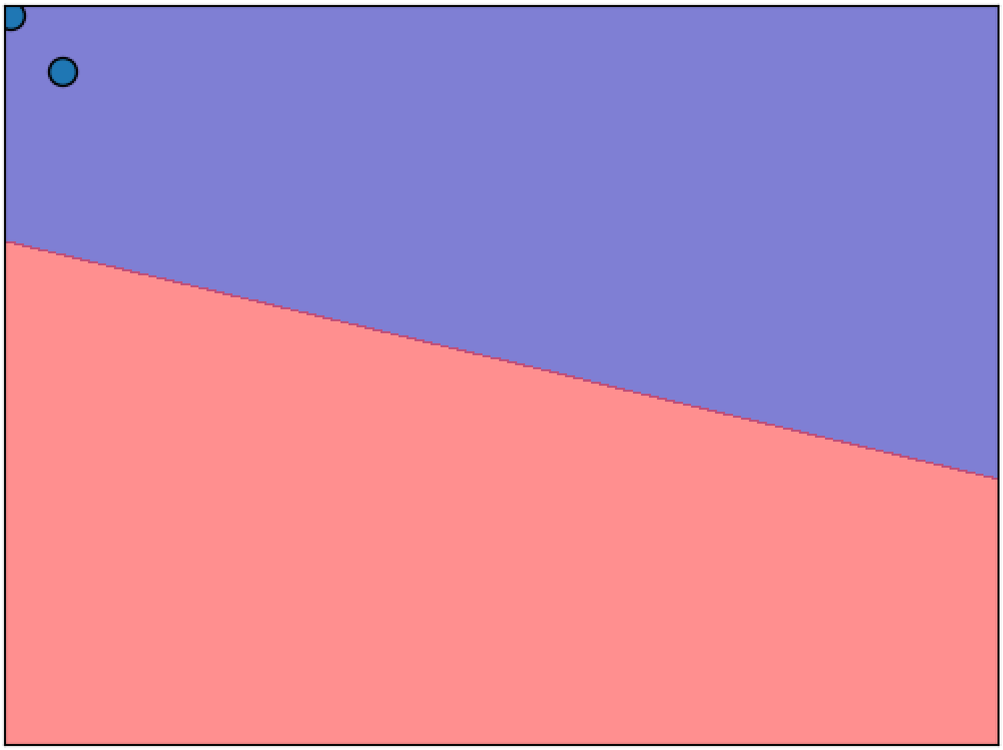
In [56]:
```
probs = lr.predict_proba(over_confident_X.to_numpy())

data_dict = {
    "y": over_confident_y,
    "y_hat": lr.predict(over_confident_X.to_numpy()).tolist(),
    "probability score (Canada)": probs[:, 0],
    "probability score (USA)": probs[:, 1],
}
pd.DataFrame(data_dict)
```

Out[56]:

| | y | y_hat | probability score (Canada) | probability score (USA) |
|---|---|---|---|---|
| **0** | USA | Canada | 0.932487 | 0.067513 |
| **1** | USA | Canada | 0.961902 | 0.038098 |

In [57]:

```
mglearn.discrete_scatter(
    over_confident_X.iloc[:, 0],
    over_confident_X.iloc[:, 1],
    over_confident_y,
    markers="o",
)
mglearn.plots.plot_2d_separator(lr, X_train.to_numpy(), fill=True, eps=
```

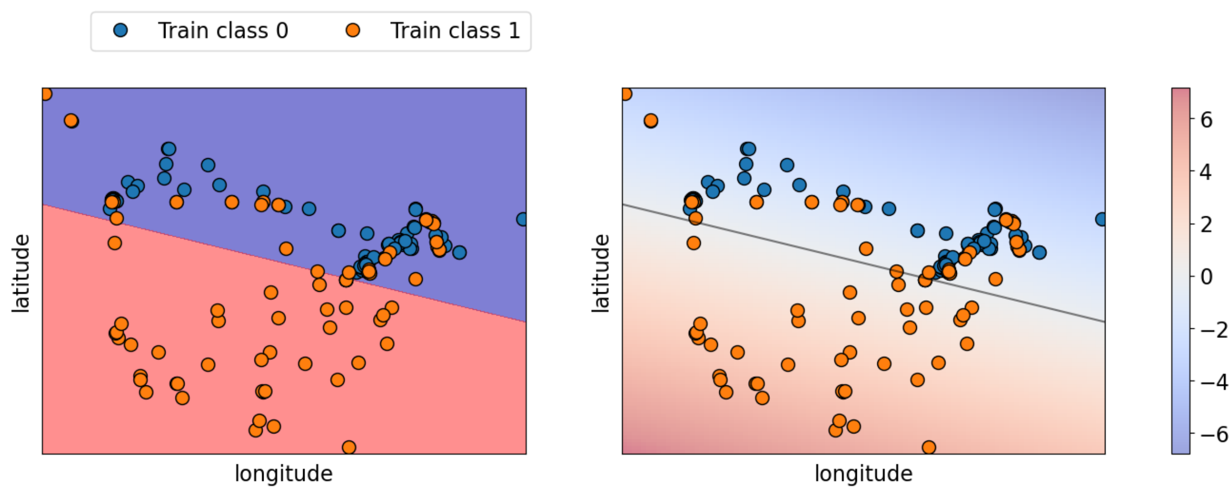


- The cities are far away from the decision boundary. So the model is pretty confident about the prediction.
- But the cities are likely to be from Alaska and our linear model is not able to capture that this part belong to the USA and not Canada.

Below we are using colour to represent prediction probabilities. If you are closer to the border, the model is less confident whereas the model is more confident about the mainland cities, which makes sense.

In [58]:

```
fig, axes = plt.subplots(1, 2, figsize=(18, 5))
from matplotlib.colors import ListedColormap

for ax in axes:
    mglearn.discrete_scatter(
        X_train.iloc[:, 0], X_train.iloc[:, 1], y_train, markers="o", a
    )
    ax.set_xlabel("longitude")
    ax.set_ylabel("latitude")

axes[0].legend(["Train class 0", "Train class 1"], ncol=2, loc=(0.1, 1.

mglearn.plots.plot_2d_separator(
    lr, X_train.to_numpy(), fill=True, eps=0.5, ax=axes[0], alpha=0.5
)
mglearn.plots.plot_2d_separator(
    lr, X_train.to_numpy(), fill=False, eps=0.5, ax=axes[1], alpha=0.5
)
scores_image = mglearn.tools.plot_2d_scores(
    lr, X_train.to_numpy(), eps=0.5, ax=axes[1], alpha=0.5, cm=plt.cm.c
)
cbar = plt.colorbar(scores_image, ax=axes.tolist())
```




Sometimes a complex model that is overfitted, tends to make more confident predictions, even if they are wrong, whereas a simpler model tends to make predictions with more uncertainty.

To summarize,

- With hard predictions, we only know the class.
- With probability scores we know how confident the model is with certain predictions, which can be useful in understanding the model better.

## ❓ ❓ Questions for you

**True/False (for practice)**

- Increasing logistic regression's `C` hyperparameter increases model complexity.
- Unlike with `Ridge` regression, coefficients are not interpretable with logistic regression.
- The raw output score can be used to calculate the probability score for a given prediction.
- For linear classifier trained on $d$ features, the decision boundary is a $d-1$-dimensional hyperparlane.
- A linear model is likely to be uncertain about the data points close to the decision boundary.
- Similar to decision trees, conceptually logistic regression should be able to work with categorical features.
- Scaling might be a good idea in the context of logistic regression.

**Zoom poll**

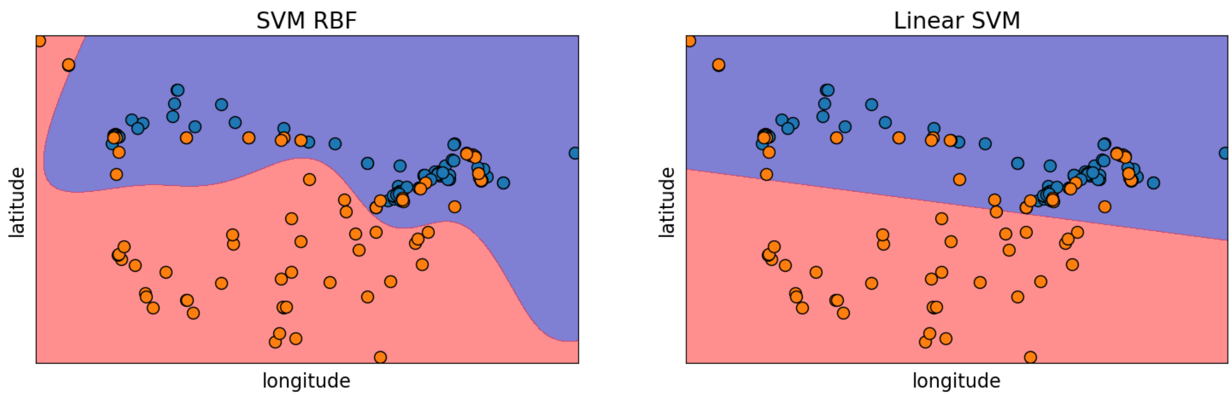
1. The intercept of a linear model has a meaning similar to the model's coefficients (T/F)
2. The snake weight linear model was trained on snakes between 0 and 200 cm of length. Should we try predicting the weight of a 250 cm long snake?
3. A positive coefficient indicates positive correlation between the feature and the target (T/F)
4. RIDGE regression helps controlling the model complexity by limiting the number of non-zero coefficients in the model (T/F)
5. Logistic regression only accepts binary inputs (T/F)
6. Changing the order of the classes in logistic regression will not affect the coefficients (T/F)
7. A larger coefficient indicates a stronger correlation (T/F)

## Linear SVM

- We have seen non-linear SVM with RBF kernel before. This is the default SVC model in `sklearn` because it tends to work better in many cases.
- There is also a linear SVM. You can pass `kernel="linear"` to create a linear SVM.

In [59]:
```
cities_df = pd.read_csv("../data/canada_usa_cities.csv")
train_df, test_df = train_test_split(cities_df, test_size=0.2, random_s
X_train, y_train = train_df.drop(columns=["country"], axis=1), train_df
X_test, y_test = test_df.drop(columns=["country"], axis=1), test_df["co
```

In [60]:
```
fig, axes = plt.subplots(1, 2, figsize=(18, 5))
from matplotlib.colors import ListedColormap

for (model, ax) in zip([SVC(gamma=0.01), SVC(kernel="linear")], axes):
    mglearn.discrete_scatter(
        X_train.iloc[:, 0].to_numpy(), X_train.iloc[:, 1].to_numpy(), y
    )
    model.fit(X_train.to_numpy(), y_train)
    ax.set_xlabel("longitude")
    ax.set_ylabel("latitude")
    mglearn.plots.plot_2d_separator(
        model, X_train.to_numpy(), fill=True, eps=0.5, ax=ax, alpha=0.5
    )

axes[0].set_title("SVM RBF")
axes[1].set_title("Linear SVM");
```




- `predict` method of linear SVM and logistic regression works the same way.
- We can get `coef_` associated with the features and `intercept_` using a Linear SVM model.

In [61]:
```
linear_svc = SVC(kernel="linear")
linear_svc.fit(X_train, y_train)
print("Model weights: %s" % (linear_svc.coef_))
print("Model intercept: %s" % (linear_svc.intercept_))
```

```
Model weights: [[-0.0195598  -0.23640124]]
Model intercept: [8.22811601]
```

In [62]:
```
lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Model weights: %s" % (lr.coef_))
print("Model intercept: %s" % (lr.intercept_))
```

```
Model weights: [[-0.04108149 -0.33683126]]
Model intercept: [10.8869838]
```

- Note that the coefficients and intercept are slightly different for logistic regression.
- This is because the `fit` for linear SVM and logistic regression are different.

## Model interpretation of linear classifiers

- One of the primary advantage of linear classifiers is their ability to interpret models.
- For example, with the sign and magnitude of learned coefficients we could answer questions such as which features are driving the prediction to which direction.

- We'll demonstrate this by training `LogisticRegression` on the famous [IMDB movie review (https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews)](https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews) dataset. The dataset is a bit large for demonstration purposes. So I am going to put a big portion of it in the test split to speed things up.

In [63]:

```
imdb_df = pd.read_csv("../data/imdb_master.csv", encoding="ISO-8859-1")
imdb_df = imdb_df[imdb_df["label"].str.startswith(("pos", "neg"))]
imdb_df.drop(["Unnamed: 0", "type", "file"], axis=1, inplace=True)
imdb_df.head()
```

Out[63]:

| | review | label |
|---|---|---|
| **0** | Once again Mr. Costner has dragged out a movie for far longer than necessary. Aside from the terrific sea rescue sequences, of which there are very few I just did not care about any of the charact... | neg |
| **1** | This is an example of why the majority of action films are the same. Generic and boring, there's really nothing worth watching here. A complete waste of the then barely-tapped talents of Ice-T and... | neg |
| **2** | First of all I hate those moronic rappers, who could'nt act if they had a gun pressed against their foreheads. All they do is curse and shoot each other and acting like clichÃ©'e version of gangst... | neg |
| **3** | Not even the Beatles could write songs everyone liked, and although Walter Hill is no mop-top he's second to none when it comes to thought provoking action movies. The nineties came and social pla... | neg |
| **4** | Brass pictures (movies is not a fitting word for them) really are somewhat brassy. Their alluring visual qualities are reminiscent of expensive high class TV commercials. But unfortunately Brass p... | neg |

Let's clean up the data a bit.

In [64]:

```
import re


def replace_tags(doc):
    doc = doc.replace("<br />", " ")
    doc = re.sub("https://\S*", "", doc)
    return doc
```

In [65]:
```
imdb_df["review_pp"] = imdb_df["review"].apply(replace_tags)
```

Are we breaking the Golden rule here?

Let's split the data and create bag of words representation.

In [66]:
```
train_df, test_df = train_test_split(imdb_df, test_size=0.9, random_sta
X_train, y_train = train_df["review_pp"], train_df["label"]
X_test, y_test = test_df["review_pp"], test_df["label"]
train_df.shape
```

Out[66]:
```
(5000, 3)
```

In [67]:
```
vec = CountVectorizer(stop_words="english", max_features=10000)
bow = vec.fit_transform(X_train)
bow
```

Out[67]:
```
<5000x10000 sparse matrix of type '<class 'numpy.int64'>'
	with 383702 stored elements in Compressed Sparse Row format>
```

## Examining the vocabulary

- The vocabulary (mapping from feature indices to actual words) can be obtained using `get_feature_names()` on the `CountVectorizer` object.

In [68]:
```
vocab = vec.get_feature_names_out()
```

In [69]:
```
vocab[0:10]  # first few words
```

Out[69]:
```
array(['00', '000', '01', '10', '100', '1000', '101', '11', '12', '13'],
      dtype=object)
```

In [70]:
```
vocab[2000:2010]  # some middle words
```

Out[70]:
```
array(['conrad', 'cons', 'conscience', 'conscious', 'consciously',
       'consciousness', 'consequence', 'consequences', 'conservative',
       'conservatory'], dtype=object)
```

In [71]:
```
vocab[::500]  # words with a step of 500
```

Out[71]:
```
array(['00', 'announcement', 'bird', 'cell', 'conrad', 'depth', 'elite',
       'finnish', 'grimy', 'illusions', 'kerr', 'maltin', 'narrates',
       'patients', 'publicity', 'reynolds', 'sfx', 'starting', 'thats',
       'vance'], dtype=object)
```

## Model building on the dataset

First let's try `DummyClassifier` on the dataset.

In [72]:
```
dummy = DummyClassifier()
scores = cross_validate(dummy, X_train, y_train, return_train_score=Tru
pd.DataFrame(scores)
```

Out[72]:

| | fit_time | score_time | test_score | train_score |
|---|---|---|---|---|
| **0** | 0.002985 | 0.002214 | 0.505 | 0.505 |
| **1** | 0.002587 | 0.001751 | 0.505 | 0.505 |
| **2** | 0.002568 | 0.001805 | 0.505 | 0.505 |
| **3** | 0.004138 | 0.002139 | 0.505 | 0.505 |
| **4** | 0.003779 | 0.001940 | 0.505 | 0.505 |

We have a balanced dataset. So the `DummyClassifier` score is around 0.5.

Now let's try logistic regression.

In [73]:
```
pipe_lr = make_pipeline(
    CountVectorizer(stop_words="english", max_features=10000),
    LogisticRegression(max_iter=1000),
)
scores = cross_validate(pipe_lr, X_train, y_train, return_train_score=T
pd.DataFrame(scores)
```

Out[73]:

| | fit_time | score_time | test_score | train_score |
|---|---|---|---|---|
| **0** | 1.354910 | 0.242920 | 0.847 | 1.0 |
| **1** | 1.171952 | 0.246972 | 0.832 | 1.0 |
| **2** | 1.238927 | 0.225187 | 0.842 | 1.0 |
| **3** | 1.142833 | 0.219543 | 0.853 | 1.0 |
| **4** | 1.294660 | 0.213265 | 0.839 | 1.0 |

Seems like we are overfitting. Let's optimize the hyperparameter `C`.

In [74]:
```
scores_dict = {
    "C": 10.0 ** np.arange(-3, 3, 1),
    "mean_train_scores": list(),
    "mean_cv_scores": list(),
}
for C in scores_dict["C"]:
    pipe_lr = make_pipeline(
        CountVectorizer(stop_words="english", max_features=10000),
        LogisticRegression(max_iter=1000, C=C),
    )
    scores = cross_validate(pipe_lr, X_train, y_train, return_train_sco
    scores_dict["mean_train_scores"].append(scores["train_score"].mean(
    scores_dict["mean_cv_scores"].append(scores["test_score"].mean())

results_df = pd.DataFrame(scores_dict)
results_df
```

Out[74]:

| | C | mean_train_scores | mean_cv_scores |
|---|---|---|---|
| **0** | 0.001 | 0.83470 | 0.7964 |
| **1** | 0.010 | 0.92265 | 0.8456 |
| **2** | 0.100 | 0.98585 | 0.8520 |
| **3** | 1.000 | 1.00000 | 0.8426 |
| **4** | 10.000 | 1.00000 | 0.8376 |
| **5** | 100.000 | 1.00000 | 0.8350 |

In [75]:
```
optimized_C = results_df["C"].iloc[np.argmax(results_df["mean_cv_scores
print(
    "The maximum validation score is %0.3f at C = %0.2f "
    % (
        np.max(results_df["mean_cv_scores"]),
        optimized_C,
    )
)
```

```
The maximum validation score is 0.852 at C = 0.10
```

Let's train a model on the full training set with the optimized hyperparameter values.

In [76]:
```
pipe_lr = make_pipeline(
    CountVectorizer(stop_words="english", max_features=10000),
    LogisticRegression(max_iter=1000, C=optimized_C),
)
pipe_lr.fit(X_train, y_train)
```

Out[76]:
```
Pipeline(steps=[('countvectorizer',
                 CountVectorizer(max_features=10000, stop_words='englis
h')),
                ('logisticregression',
                 LogisticRegression(C=0.1, max_iter=1000))])
```

**In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.**
**On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.**

## Examining learned coefficients

- The learned coefficients are exposed by the `coef_` attribute of [LogisticRegression (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) object.

In [77]:
```
feature_names = np.array(pipe_lr.named_steps["countvectorizer"].get_fea
coeffs = pipe_lr.named_steps["logisticregression"].coef_.flatten()
```

In [78]:
```
word_coeff_df = pd.DataFrame(coeffs, index=feature_names, columns=["Coe
word_coeff_df
```

Out[78]:

| | Coefficient |
|---|---|
| **00** | -0.074949 |
| **000** | -0.083893 |
| **01** | -0.034402 |
| **10** | 0.056493 |
| **100** | 0.041633 |
| **...** | ... |
| **zoom** | -0.013299 |
| **zooms** | -0.022139 |
| **zorak** | 0.021878 |
| **zorro** | 0.130075 |
| **â½** | 0.012649 |

10000 rows × 1 columns

- Let's sort the coefficients in descending order.
- Interpretation
  - if $w_j > 0$ then increasing $x_{ij}$ moves us toward predicting $+1$.
  - if $w_j < 0$ then increasing $x_{ij}$ moves us toward predicting $-1$.

In [79]:
```
word_coeff_df.sort_values(by="Coefficient", ascending=False)
```

Out[79]:

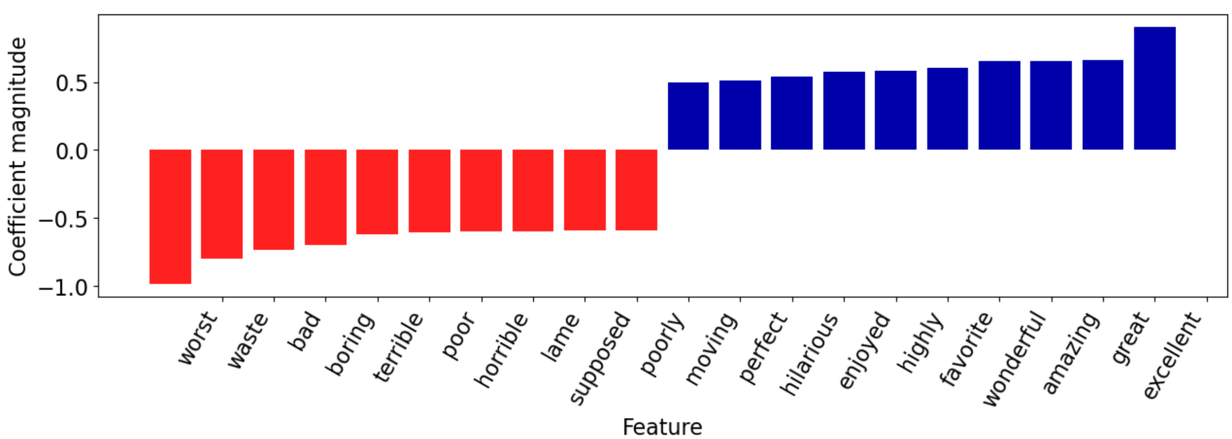
| | Coefficient |
|---|---|
| **excellent** | 0.903484 |
| **great** | 0.659922 |
| **amazing** | 0.653301 |
| **wonderful** | 0.651763 |
| **favorite** | 0.607887 |
| **...** | ... |
| **terrible** | -0.621695 |
| **boring** | -0.701030 |
| **bad** | -0.736608 |
| **waste** | -0.799353 |
| **worst** | -0.986970 |

10000 rows × 1 columns

- The coefficients make sense!

Let's visualize the top 10 features.

In [80]:
```
mglearn.tools.visualize_coefficients(coeffs, feature_names, n_top_featu
```




Let's explore prediction of the following new review.

In [81]:
```
fake_review = "It got a bit boring at times but the direction was excel
```

In [82]:
```
feat_vec = pipe_lr.named_steps["countvectorizer"].transform([fake_revie
```

In [83]:
```
feat_vec
```

Out[83]:
```
<1x10000 sparse matrix of type '<class 'numpy.int64'>'
	with 12 stored elements in Compressed Sparse Row format>
```

Let's get prediction probability scores of the fake review.

In [84]:
```
pipe_lr.predict_proba([fake_review])
```

Out[84]:
```
array([[0.1718113, 0.8281887]])
```

The model is 82% confident that it's a positive review.

In [85]:
```
pipe_lr.predict([fake_review])[0]
```

Out[85]:
```
'pos'
```

We can find which of the vocabulary words are present in this review:

In [86]:
```
feat_vec.toarray().ravel().astype(bool)
```

Out[86]:
```
array([False, False, False, ..., False, False, False])
```

In [87]:
```
words_in_ex = feat_vec.toarray().ravel().astype(bool)
words_in_ex
```

Out[87]:
```
array([False, False, False, ..., False, False, False])
```

How many of the words are in this review?

In [88]:
```
np.sum(words_in_ex)
```

Out[88]:
```
12
```

In [89]:
```
np.array(feature_names)[words_in_ex]
```

Out[89]:
```
array(['acting', 'bit', 'boring', 'direction', 'enjoyed', 'excellent',
       'flawless', 'got', 'highly', 'movie', 'overall', 'times'],
      dtype=object)
```

In [90]:

```
ex_df = pd.DataFrame(
    data=coeffs[words_in_ex],
    index=np.array(feature_names)[words_in_ex],
    columns=["Coefficient"],
)
ex_df
```

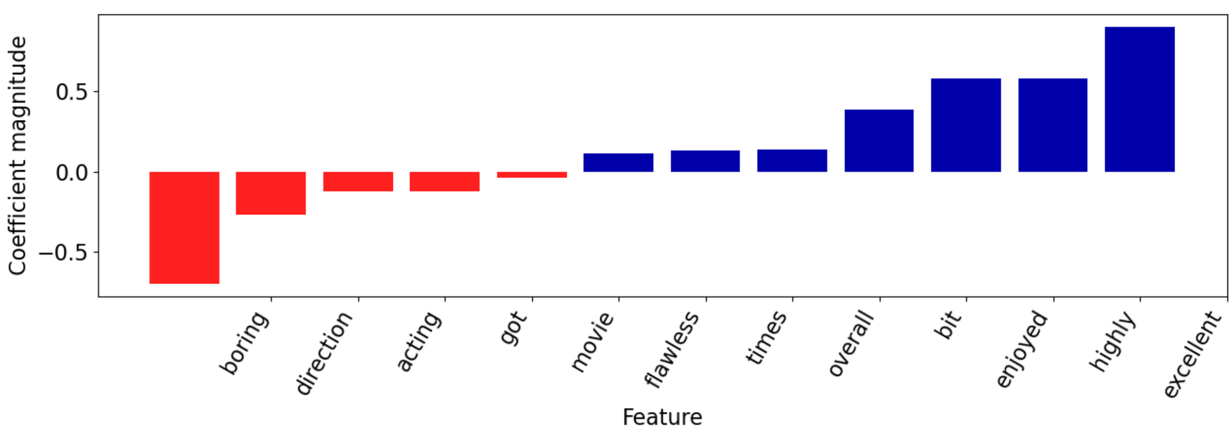
Out[90]:

| | Coefficient |
|---|---|
| **acting** | -0.126498 |
| **bit** | 0.390053 |
| **boring** | -0.701030 |
| **direction** | -0.268316 |
| **enjoyed** | 0.578879 |
| **excellent** | 0.903484 |
| **flawless** | 0.113743 |
| **got** | -0.122759 |
| **highly** | 0.582012 |
| **movie** | -0.037942 |
| **overall** | 0.136288 |
| **times** | 0.133895 |

Let's visualize how the words with positive and negative coefficients are driving the hard prediction.

In [91]:

```
mglearn.tools.visualize_coefficients(
    coeffs[words_in_ex], np.array(feature_names)[words_in_ex], n_top_fe
)
```



In [92]:
```
def plot_coeff_example(feat_vect, coeffs, feature_names):
    words_in_ex = feat_vec.toarray().ravel().astype(bool)

    ex_df = pd.DataFrame(
        data=coeffs[words_in_ex],
        index=np.array(feature_names)[words_in_ex],
        columns=["Coefficient"],
    )
    return ex_df
```

## Most positive review

- Remember that you can look at the probabilities (confidence) of the classifier's prediction using the `model.predict_proba` method.
- Can we find the messages where our classifier is most confident or least confident?

In [93]:
```
pos_probs = pipe_lr.predict_proba(X_train)[
    :, 1
]  # only get probabilities associated with pos class
pos_probs
```

Out[93]:
```
array([0.95205899, 0.83301769, 0.9093526 , ..., 0.89247531, 0.05736279,
       0.79360853])
```

Let's get the index of the example where the classifier is most confident (highest `predict_proba` score for positive).

In [94]:
```
most_positive = np.argmax(pos_probs)
```

In [95]:
```
X_train.iloc[most_positive]
```

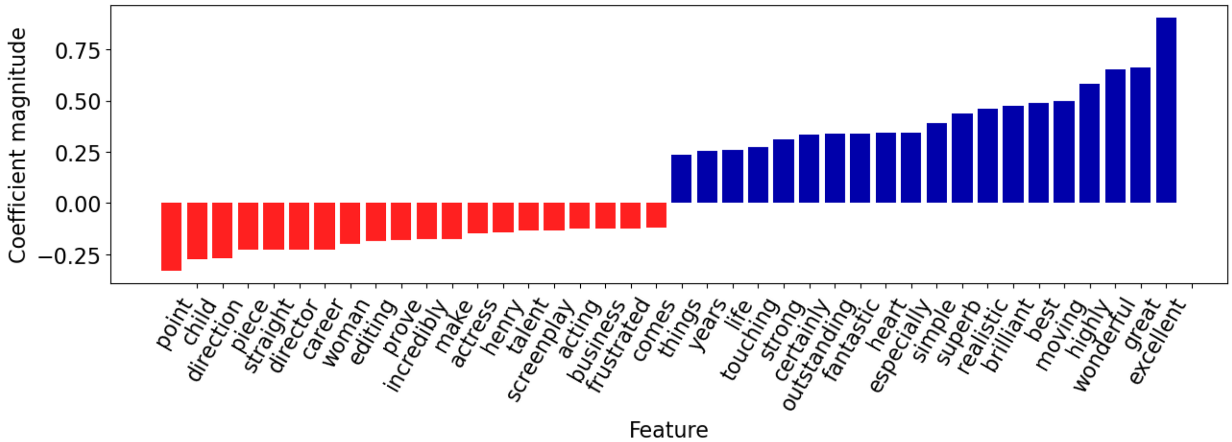
Out[95]:
```
'Moving beyond words is this heart breaking story of a divorce which resu
lts in a tragic custody battle over a seven year old boy.  One of "Kramer
v. Kramer\'s" great strengths is its screenwriter director Robert Benton,
who has marvellously adapted Avery Corman\'s novel to the big screen. He
keeps things beautifully simple and most realistic, while delivering all
the drama straight from the heart. His talent for telling emotional tales
like this was to prove itself again with "Places in the Heart", where he
showed, as in "Kramer v. Kramer", that he has a natural ability for worki
ng with children.  The picture\'s other strong point is the splendid acti
ng which deservedly received four of the film\'s nine Academy Award nomin
ations, two of them walking away winners. One of those was Dustin Hoffman
(Best Actor), who is superb as frustrated business man Ted Kramer, a man
who has forgotten that his wife is a person. As said wife Joanne, Meryl S
treep claimed the supporting actress Oscar for a strong, sensitive portra
yal of a woman who had lost herself in eight years of marriage. Also nomi
nated was Jane Alexander for her fantastic turn as the Kramer\'s good fri
end Margaret. Final word in the acting stakes must go to young Justin Hen
ry, whose incredibly moving performance will find you choking back tears
again and again, and a thoroughly deserved Oscar nomination came his way.
Brilliant also is Nestor Almendros\' cinematography and Jerry Greenberg
\'s timely editing, while musically Henry Purcell\'s classical piece is u
sed to effect.  Truly this is a touching story of how a father and son co
me to depend on each other when their wife and mother leaves. They grow t
ogether, come to know each other and form an entirely new and wonderful r
elationship. Ted finds himself with new responsibilities and a new outloo
k on life, and slowly comes to realise why Joanne had to go.  Certainly i
f nothing else, "Kramer v. Kramer" demonstrates that nobody wins when it
comes to a custody battle over a young child, especially not the child hi
mself.  Saturday, June 10, 1995 - T.V.  Strong drama from Avery Corman\'s
novel about the heartache of a custody battle between estranged parents w
ho both feel they have the child\'s best interests at heart. Aside from a
superb screenplay and amazingly controlled direction, both from Robert Be
nton, it\'s the superlative cast that make this picture such a winner.  H
offman is brilliant as Ted Kramer, the man torn between his toppling care
er and the son whom he desperately wants to keep. Excellent too is Streep
as the woman lost in eight years of marriage who had to get out before sh
e faded to nothing as a person. In support of these two is a very strong
Jane Alexander as mutual friend Margaret, an outstanding Justin Henry as
the boy caught in the middle, and a top cast of extras.  This highly emot
ional, heart rending drama more than deserved it\'s 1979 Academy Awards f
or best film, best actor (Hoffman) and best supporting actress (Streep).
Wednesday, February 28, 1996 - T.V.'
```

In [96]:
```
print("True target: %s\n" % (y_train.iloc[most_positive]))
print("Predicted target: %s\n" % (pipe_lr.predict(X_train.iloc[[most_po
print("Prediction probability: %0.4f" % (pos_probs[most_positive]))
```

```
True target: pos

Predicted target: pos

Prediction probability: 1.0000
```

Let's examine the features associated with the review.

In [97]:
```
feat_vec = pipe_lr.named_steps["countvectorizer"].transform(
    X_train.iloc[[most_positive]]
)
words_in_ex = feat_vec.toarray().ravel().astype(bool)
mglearn.tools.visualize_coefficients(
    coeffs[words_in_ex], np.array(feature_names)[words_in_ex], n_top_fe
)
```




The review has both positive and negative words but the words with **positive** coefficients win in this case!

## Most negative review

In [98]:
```
neg_probs = pipe_lr.predict_proba(X_train)[
    :, 0
]  # only get probabilities associated with pos class
neg_probs
```

Out[98]:
```
array([0.04794101, 0.16698231, 0.0906474 , ..., 0.10752469, 0.94263721,
       0.20639147])
```

In [99]:
```
most_negative = np.argmax(neg_probs)
```

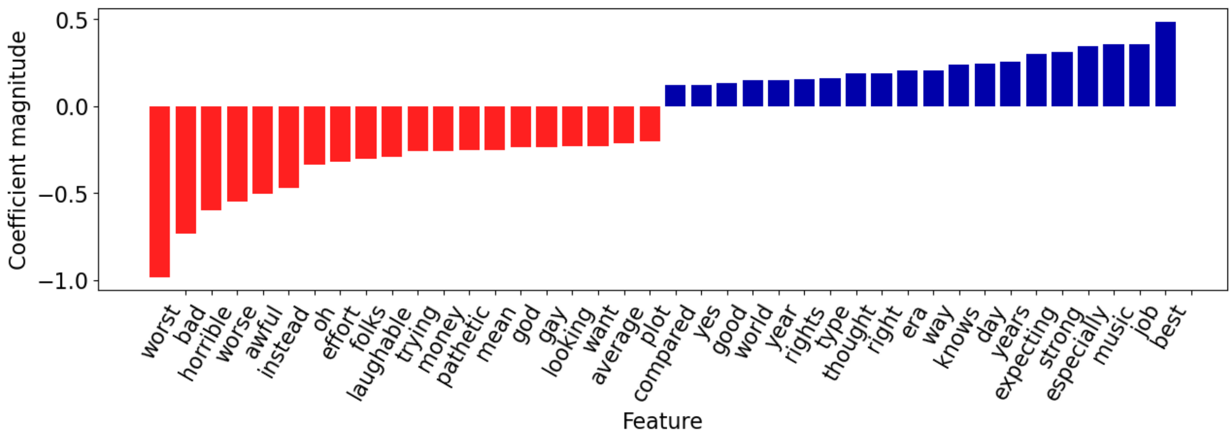
In [100]:

```
print("Review: %s\n" % (X_train.iloc[[most_negative]]))
print("True target: %s\n" % (y_train.iloc[most_negative]))
print("Predicted target: %s\n" % (pipe_lr.predict(X_train.iloc[[most_ne
print("Prediction probability: %0.4f" % (pos_probs[most_negative]))
```

```
Review: 36555    I made the big mistake of actually watching this whole m
ovie a few nights ago. God I'm still trying to recover. This movie does n
ot even deserve a 1.4 average. IMDb needs to have 0 vote ratings po...
Name: review_pp, dtype: object

True target: neg

Predicted target: neg

Prediction probability: 0.0000
```

In [101]:

```
feat_vec = pipe_lr.named_steps["countvectorizer"].transform(
    X_train.iloc[[most_negative]]
)
words_in_ex = feat_vec.toarray().ravel().astype(bool)
mglearn.tools.visualize_coefficients(
    coeffs[words_in_ex], np.array(feature_names)[words_in_ex], n_top_fe
)
```




The review has both positive and negative words but the words with negative coefficients win in this case!

## ? ? Questions for you

#### Question for you to ponder on

- Is it possible to identify most important features using $k$-NNs? What about decision trees?

# Summary of linear models

- Linear regression is a linear model for regression whereas logistic regression is a linear model for classification.
- Both these models learn one coefficient per feature, plus an intercept.

### Main hyperparameters

- The main hyperparameter is the "regularization" hyperparameter controlling the fundamental tradeoff.
  - Logistic Regression: `C`
  - Linear SVM: `C`
  - Ridge: `alpha`

### Interpretation of coefficients in linear models

- the $j$th coefficient tells us how feature $j$ affects the prediction
- if $w_j > 0$ then increasing $x_{ij}$ moves us toward predicting $+1$
- if $w_j < 0$ then increasing $x_{ij}$ moves us toward prediction $-1$
- if $w_j == 0$ then the feature is not used in making a prediction

### Strengths of linear models

- Fast to train and predict
- Scale to large datasets and work well with sparse data
- Relatively easy to understand and interpret the predictions
- Perform well when there is a large number of features

### Limitations of linear models

- Is your data "linearly separable"? Can you draw a hyperplane between these datapoints that separates them with 0 error.
  - If the training examples can be separated by a linear decision rule, they are **linearly separable**.

A few questions you might be thinking about

- How often the real-life data is linearly separable?
- Is the following XOR function linearly separable?

| $x_1$ | $x_2$ | **target** |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |